

Getting Started with HI-TECH C for PIC10/12/16 MCUs, Microchip PICDEM™ 2 PLUS Board and MPLAB® ICD 2

Produced on: June 14, 2011

Australian Design Centre
45 Colebard Street West
Acacia Ridge QLD 4110
Australia

web: http://www.htsoft.com

Chapter 1

Introduction

This guide is intended to introduce you to HI-TECH C for PIC10/12/16 MCUs and its use in the Microchip MPLAB IDE. Code will be produced for the Microchip PICDEM 2 PLUS demo board, running a PIC16F877A MCU device. Programming will be performed via a Microchip MPLAB ICD 2 in-circuit debugger. Code will be written to flash the 4 LEDs on the board.

This guide will illustrate the following points.

- Setting up MPLAB IDE projects to use the HI-TECH C compiler;
- Compiling for the Microchip MPLAB ICD 2 debugger;
- Becoming familiar with HI-TECH C source code and the HI-TECH C Compiler for PIC10/12/16 MCUs;
- Accessing special function registers;
- Writing interrupt code; and
- Defining code in more than one source file.

1.1 Before you Begin

The following chapters assume that you have already installed the Microchip MPLAB IDE.

Microchip use special drivers to communicate with the MPLAB ICD 2. Microchip have documentation relating to installation of the IDE and drivers.

You will also need to have installed the HI-TECH C Compiler for the PIC10/12/16 MCUs. You can use the fully licensed, evaluation or LITE (free) version of this compiler with this guide. The

compiler should be installed after first installing MPLAB IDE. The compiler installer will automatically install the Universal Toolsuite plugin for MPLAB IDE that allows the IDE to drive the compiler.

You will need to follow the Microchip documentation to set up the hardware connections from your PC to the MPLAB ICD 2, and from the MPLAB ICD 2 to the PICDEM 2 board. Ensure that the 16F877A device is installed in the PICDEM 2 board.

Chapter 2

Setting up a Project in MPLAB IDE

Begin by running MPLAB IDE.

The first step to using MPLAB IDE is to open a project. We don't have an existing project to open so we will need to create a new one. The easiest way to do that is use the Project Wizard in MPLAB IDE. Select the Project Wizard menu item from the Project pull-down menu as illustrated in Figure 2.1.

In the dialog that opens, click `Next` to move to the chip selection dialog. Select a `PIC16F877A` device as shown in Figure 2.2 then click `Next`.

The next dialog is where you get to specify the toolsuite associated with the project. You are not associating the project with any one particular compiler, but rather specifying that you will be using one of the HI-TECH C compilers for Microchip devices. Once the project has been created, you

Figure 2.1: Initiating the MPLAB IDE Project Wizard

Figure 2.2: Selecting the target device

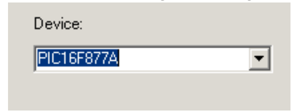
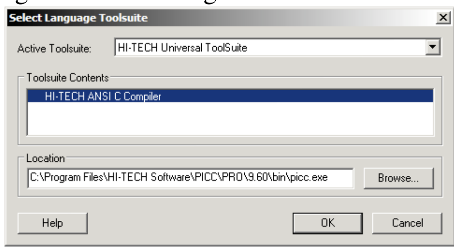


Figure 2.3: Selecting the HI-TECH toolsuite



can specify the compiler version that you will use, and you can change the compiler version without having to re-create the project. To use the HI-TECH Family of compilers you must select `HI-TECH Universal Toolsuite` as the `Active Toolsuite` in this dialog, as shown in Figure 2.3.

Notice that the `Toolsuite Contents` area shows the generic name `HI-TECH ANSI C Compiler`. Notice also the path shown in the `Location` area. This path is not important and you do not need to edit it to point to the compiler location. The Universal Toolsuite determines the location of the installed compilers via other means. Click `Next`.

In the next dialog you can specify the name and location of the project you are creating. Click `Next` after deciding on something appropriate.

Lastly the Project Wizard is asking to see if you would like any existing files to be added to the project. We are going to create a small amount of code from scratch so we do not wish to add any files. Click `Next` to skip this dialog and show the project summary. If everything looks okay in the summary click `Finish` and we are done.

Being the good engineers we are, we are going to create a small skeleton program and ensure that we can compile that before getting too involved using the features of the PICDEM 2 PLUS board. You should aim to do this every time you create a new project.

In MPLAB IDE, select the `New` menu item from the `File` menu, or use any of MPLAB IDE's short cuts methods to create a new file. In the empty file window that opens, copy the code shown in Figure 2.4, and paste it into the window. (If your PDF browser does not co-operate in this aim, you may need to type the code in — rest assured there is not much code that we need for this guide.)

Figure 2.4: A small test program to ensure everything is setup

```
#include <htc.h>

void main(void)
{
}
```

This is all we need to create a complete C program. It will not do very much at all, apart from start and end almost immediately, however every program you write will need a function called `main`. The include file called `htc.h` is not actually needed since we have no code inside `main`, however you will need it for virtually every file you write, and so we will put it in now to help make this a habit.

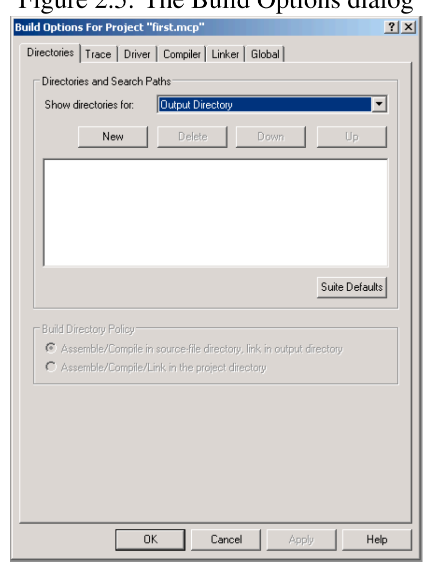
Let's now save the file and add it to our project. Click the `Save` menu item under the `File` menu. I suggest you save the file in the same directory as the project file so everything is together. (If you can't remember where you saved the project, click on the `Project` menu and then place the your mouse pointer on the `Close` menu item. This will open a side menu that will show you the full path to the project that is currently open. Do *not* click on the side menu item as that will actually close this project.) As this is a C source file, make sure you use the extension `.c` with the file name. Next right-click in the editor window that contains our code and select `Add to Project`, about three quarters the way down this menu.

To ensure that the file was correctly added, click the `View` menu, and select `Project`. This will open a window which shows an overview of the project. In this window you will see several folder icons. Under the `Source Files` icon you should see the name of the file listed next to a file icon. If you close the editor window containing our code, you can re-open it by double-clicking the file icon associated with the file in the `Project` window.

Select the `Project` menu. About half way down is a menu item `Build Options...`. If you place your mouse pointer over this menu item a side menu will appear that has an entry showing the name of file you added to the project and another called `Project`. Click on the `Project` item. A dialog will open, as shown in Figure 2.5. This is the dialog that contains the options for the compiler. This dialog is managed by the toolsuite that you have selected — in this case, the HI-TECH Universal Toolsuite.

We must confirm in the project settings that the correct compiler is associated with this project. Select the `Driver` tab of the `Build Options` dialog. That will show settings similar to those shown in Figure 2.6. At the top is shown a list of `Available Drivers`. You may only have

Figure 2.5: The Build Options dialog



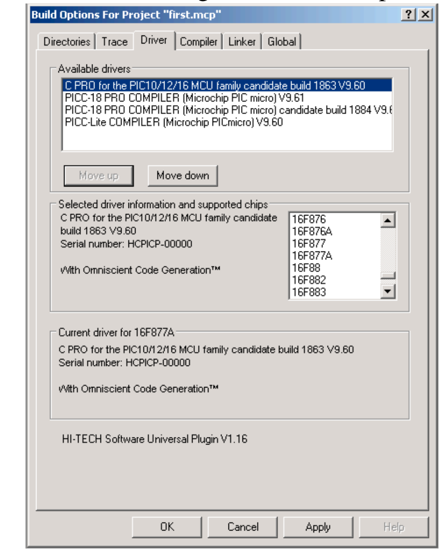
one HI-TECH C compiler supporting Microchip installed, in which case you will see its name only listed. However if you have other HI-TECH C compilers supporting Microchip installed they will all be listed in this dialog. You may have two different versions of the same compiler installed, for instance. Select the compiler you wish to use and if it is not already, click the `Move up` button to place it at the top of the list.[1] If you are not sure which compiler you wish to use, select it in the list, then look at the information in the area marked `Selected driver information and supported chips`. This area gives information about the compiler and shows in the scrollable list those chips that the compiler supports. You do not need to select the device you are using from this list — the list merely indicates the supported devices. The target device was specified when we created the project and can be changed via a different MPLAB dialog. Click `OK` when you have moved the appropriate driver to the top of the compiler list.

We should, at this point, also change one option that we will need if you intend to debug code on the MPLAB ICD 2 later on in this guide.[2] Select the the `Linker` tab of the `Build Options`

[1] If you want the real explanation, the compiler that is executed is the first one in the list that can compile for the device you have selected. So it doesn't actually have to be at the top. But to be absolutely sure, we will place it at the top of the list.

[2] This option is not required if you only intend to use the MPLAB ICD 2 as a programmer. When used as a debugger, the MPLAB ICD 2 allows the setting breakpoints and examination of variable contents etc.

Figure 2.6: Driver settings in the Build Options dialog



dialog. We need to tell the compiler that we are going to be using the MPLAB ICD 2 debugger. This is necessary as the MPLAB ICD 2 uses memory resources that might otherwise be used by ordinary program code. Select `MPLAB ICD 2` from the `Debugger` combo box widget. Click the `OK` button to leave the Build Options dialog.

At this point it would be appropriate to save the project. Select the `Save Project` menu item from the `Project` menu.

Now let's compile our small test code to ensure that MPLAB IDE and HI-TECH C are properly installed. Select `Rebuild` from the `Project` menu, or choose any of MPLAB IDE's short cuts to build the project — you can, for instance, click on the toolbar button that shows the HI-TECH "ball and stick" logo, as shown in the centre of Figure 2.7. You will notice that the Project menu has two items: `Build` and `Rebuild`. MPLAB IDE uses a two-pass compilation process — one to generate intermediate files from all the source files; the other to perform the code generation and link steps. The `Build` menu item action only processes those source files that have changed since the last build, then performs the code generation and link step. Rebuiding a project will always process every source file in the project, regardless of whether they have changed. If in doubt, use `Rebuild`. The HI-TECH Build buttons are linked to the `Build` menu item.

The program should now be compiled. You will see a window open showing content similar to

Figure 2.7: The HI-TECH build buttons

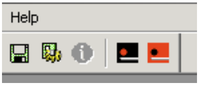
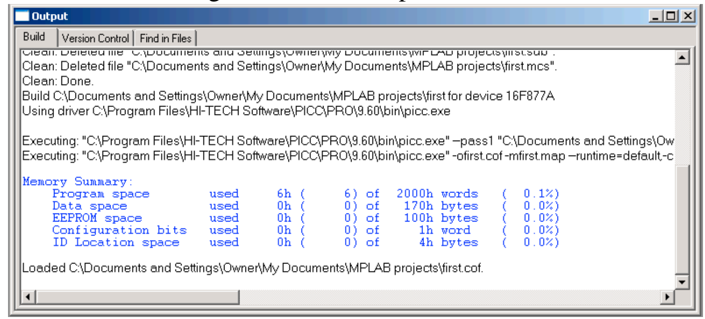


Figure 2.8: The Output window



that shown in Figure 2.8. This shows the steps both MPLAB IDE and the compiler took to build the project. The lines starting with `Executing:` show the command lines passed to the compiler command-line driver, `PICC`, that were used to actually build. Note that there are two commands: one for the only source file the project contains; the other for the code generation and link step. As more files are added to the project, more command lines will be issued by MPLAB IDE. The command lines to the compiler driver are necessarily quite long.

The compiler has produced a memory summary and there is no message indicating that the build failed, so we have successfully compiled the project.

If there are errors they will be printed in `Build` tab of this window. You can double-click each error message and MPLAB IDE will show you the offending line of code, where possible. If you do get errors, check that the program is what is contained in this document.

Chapter 3

Coding

Now that we have a project that can be compiled, we can now concentrate on developing a simple application. We will write code that flashes the four LEDs on the PICDEM 2 PLUS board. We shall have the LEDs count up in binary and initially use a software delay loop to slow down the flashing so that we can actually see the LEDs blink.

We will now flesh out our simple program. We will add:

- a macro that sets up the PIC® MCU's configuration bits;
- a function that initializes the IO port that is connected to the LEDs; and
- code to our `main` function that writes a incrementing value to the LEDs

The new complete program is shown below in Figure 3.1. It also includes the `htc.h` header file, and has a `main` function, although this function now actually contains some code. You can either replace the entire contents of the C source file we have already created with the content of the figure below, or just copy in the bits that have changed.

Save the file and then build the file as we have done previously. Again, in the Output window you should see the memory summary produced by the compiler.

Ensure at this point that you have the MPLAB ICD 2 correctly connected to your PC, that you have the MPLAB ICD 2 connected to the PICDEM 2 PLUS board, and that power is applied to the development board. From the `Debugger` menu, select `Tool`, and from the side menu, select `MPLAB ICD 2`. This tells MPLAB IDE that we will be using the MPLAB ICD 2 as the debugger. Depending on the MPLAB configuration, it may auto-connect to the MPLAB ICD 2. The `Output` window will be displayed and the connection status shown in the MPLAB ICD 2 tab. If connection

Figure 3.1: Code to flash the LEDs

```
#include <htc.h>
__CONFIG(FOSC_XT & WDTE_OFF& PWRTE_OFF & BOREN_OFF &
    LVP_ON & WRT_OFF & DEBUG_ON & CPD_OFF & CP_OFF);
void init(void)
{
        //  port directions: 1=input, 0=output
        TRISB   = 0b00000000;
}
char counter;
void main(void)
{
        counter = 0;
        init();
        while (1){
                PORTB = counter;
                _delay(10000);
                counter++;
        }
}
```

is not automatically established, select `Connect` from the `Debugger` menu. Once connection has been established, you should see a message similar to the following appear in the `Output` window.

```
Connecting to MPLAB ICD 2
...Connected
Setting Vdd source to MPLAB ICD 2
Target Device PIC16F877A found, revision = b4
...Reading ICD Product ID
Running ICD Self Test
...Passed
MPLAB ICD 2 Ready
```

You can then program the device by selecting the `Program` menu item in the `Debugger` menu. After the program has been downloaded, you should see the message:

```
...Programming succeeded
```

You can now run the code by selecting the `Run` menu item from the `Debugger` menu. You will quickly want to learn the keyboard or toolbar shortcut for the debugger commands. You should see the four LEDs slowly counting up in binary. Feel free to experiment with setting breakpoints and watching the contents of variables when you run the program. Refer to the Microchip documentation and on-line help for more details.

Chapter 4

Understanding the Code

Now that you have LEDs flashing on your board, let's look at the code we are running in more detail.

The statement that begins with __CONFIG, after the header file include directive, is a special macro that is defined by the code included with htc.h. (And from that you should note that the htc.h file must be included first before you try to use the macro.) This macro takes one argument. The argument represents a value that will be programmed into the configuration bits of the device you are using. These bits control the fundamental operation of the PIC MCU, like whether the watchdog timer will be enabled, what the source of the MCU clock will be, and whether code is protected, etc. Special macros representing the states of the configuration bits are also defined in the code included by htc.h.

The code in the function init sets up the peripherals we intend to use. In this case, we only used one peripheral: port B. The statement in this function ensures that the port behaves as we want it to — specifically that port bits 0 through to 3 are outputs; the remainder are inputs. You will notice that the compiler allows you to enter constant as binary values using the prefix 0b.

You will notice in the function init, and in the function main, that there are statements that write to variables called TRISB and PORTB. If you are familiar with the 16F877A device, you will realize that it has special function registers (SFRs) with the same name. Accessing these variables does indeed access the SFRs. Here is how this works.

In the code included with htc.h, are definitions for all the SFRs that a particular device has. These definitions are for what we call *absolute variables*, which use a special construct to place the variable over the top of the memory-mapped SFR. The definition for PORTB, for example, on this device looks like:

```
static volatile unsigned char PORTB @ 0x06;
```

which defined PORTB as an unsigned char variable that will be placed at address 6, which is

the same address used by the register with the same name. Apart from being placed at a particular address, `PORTB` is an ordinary C variable and since you know already how to use variables in expressions and statements, you do not need to learn any special syntax to access SFRs. The `volatile` keyword, along with `static`, has a standard meaning in the ANSI C language — ensure you know what it means as it is very important in the embedded programming world. We look at the `volatile` qualifier later in this guide.

The other special code in this program is the use of a delay. Notice in the `main` function there is a call to what looks like a function called `_delay` (note the leading *underscore* character), however you will not find the definition of this function in the program we have just compiled. This is a special identifier for an in-line function that is expanded by the compiler.

The argument to this function is the number of instruction cycles that will be executed, thus forming a delay. So in the code we have just executed, a delay of 10000 instruction cycles is placed in the counting loop. Without this delay the LEDs would flash so fast that their switching would not be apparent to the human eye.

Chapter 5

Going a Little Bit Further

Let's make our program a little more advanced and introduce interrupt functions in the process.

What we will do is remove the in-line delay routine, and use timer 1 on the 16F877A device to delay for us. We will also have the code spread into two C source files and have these files share a variable.

Edit you existing C source file so that your `init` function looks like that shown in Figure 5.1.

This code ensures that the port is set up as before, but also enabled timer 1. This timer has also been configured so that it will trigger an interrupt when it overflows.

Also modify your `main` function so it looks like that shown in Figure 5.2.

You can see in `main` that the delay loop was removed, and the increment of the counter was removed. We have used another macro which is defined when you include `htc.h` that enables interrupts, `ei`. Although the `init` function configures the timer to use interrupts, it does not actually

Figure 5.1: Our new `init` function

```
void init(void)
{
        PIE1    = 0b00000001;
        INTCON  = 0b01000000;
        OPTION  = 0b10000000;
        TRISB   = 0b00000000;
        T1CON   = 0b00110101;
}
```

Figure 5.2: Our new `main` function

```
#include <htc.h>
volatile char counter;
void main(void) {
        counter = 0;
        init();
        ei();
        while (1){
                PORTB = counter;
        }
}
```

enable the interrupts. Enabling interrupts before everything is setup can be disastrous, so the actual command to enable these must be placed at the appropriate place by the programmer.

Note also we have qualified the variable `counter` as `volatile`. This keyword tells the compiler that the contents of this variable may change, even though the function `main` (or any of the functions called by `main`) do not write to it. However the interrupt routine writes to it, and interrupts are not predictable and can occur at any time during execution of the main-line code. In other words, its value could change unexpected as far as the main-line code is concerned. With the `volatile` keyword in place, the compiler may access and store this variable differently so that the content of the variable is always accurately used in expressions. You should also use this qualifier for any variables that map over hardware registers that can be written to by the hardware. This is the case for the absolute variables we define in the code included by `htc.h` that map over the SFRs.

We need one more thing: the code to execute when an interrupt occurs. HI-TECH C allows you to create a special interrupt function that will be automatically linked in to the interrupt vector. In this example, we are going to place this function in a separate file to allow us to see how multiple files can be used in a project, although this is not a requirement.

In MPLAB IDE, create a new source file by selecting the `New` menu item from the `File` menu. Once open, paste in the code shown in Figure 5.3.

This code defines a function called `my_isr`. Notice the `interrupt` qualifier that actually tells the compiler that this function is to be linked to the interrupt vector. As there is only one interrupt vector on a 16F877A device, there is no other information required by the compiler. The function can have any valid C identifier as its name — you don't need to be as uncreative as I am.

The initialization of the timer in the `init` function will start timer 1 counting – and as slowly as possible. Once it overflows, an interrupt will be generated and we will end up executing code from our interrupt function. In this function, we need to increment the counter that contains the value we

Figure 5.3: Our interrupt function

```
#include <htc.h>
extern volatile char counter;
void interrupt my_isr(void){
        if((TMR1IE)&&(TMR1IF)){
                counter++;
                TMR1IF=0;
        }
}
```

will write to the LEDs in the `main` function.

But there is one slight catch, though. Although the PIC midrange devices only have one interrupt vector (hence one interrupt function) there are many sources of interrupt. So we must first ensure that the reason we are in the interrupt routine is because of the timer overflowing. This is the reason for the `if` statement inside the interrupt function. It checks to ensure that both the timer 1 interrupt flag, `TMR1IF`, (which means that timer 1 has overflowed) and that the timer 1 interrupt enable, `TMR1IE`, is set before incrementing the counter. You must remember that timers may continue to count and overflow even if you have disabled the interrupt associated with them. Typically you have more than one interrupt source in your programs so you need to ensure that you know which source triggered the interrupt.

Notice also in the interrupt function that the timer 1 interrupt flag that was set when the interrupt was generated is cleared. The PIC MCU datasheet indicates that this flag must be cleared by software.

Remember that the variable called `counter` was defined in our first source file, but we are incrementing it in this source file. We need to tell the compiler that the variable `counter`, that we are accessing here, is defined in another file. We do this by placing the declaration:

```
extern volatile char counter;
```

towards the top of the file, before counter is actually used. The compiler will know from this what type the variable is, but will not actually define memory for it. It is the `extern` keyword that indicates this statement is a declaration. You will notice it looks identical to the definition in our first source file, but for the `extern` added to the front.

Take note of the terminology being used here: a definition indicates the type of a variable and ensures memory is allocated for the variable; a declaration only indicates the type of a variable. There can only be one definition for a variable, but you can have as many declarations as you want, in as many files as you want.

Notice also that the interrupt routine is kept as small as possible. All we do is update the counter. The writing of the counter to the LEDs is performed back in our `main` function. It is desirable to have small interrupt functions as this leads to better performance in systems that must respond rapidly to events.

Compile the program, and download it to the MPLAB ICD 2 as you did in the previous example. When you run the program, it should result in the same LEDs flashing, although their speed may be a little different.